Packet-less traffic analysis using Wireshark

Luca Deri <deri@ntop.org>, @lucaderi
Samuele Sabella <sabella@ntop.org>
ntop

About Us

- Luca is the founder of the ntop project that develops open source network traffic monitoring applications. All code is available at https://github.com/ntop
- Samuele is an undergraduate student at the Computer Science Department of the University of Pisa. His interests include networking and machine learning. He's an ntop team member.
- ntop is a community: http://t.me/ntop_community
- We are part of the Intel Innovator program.

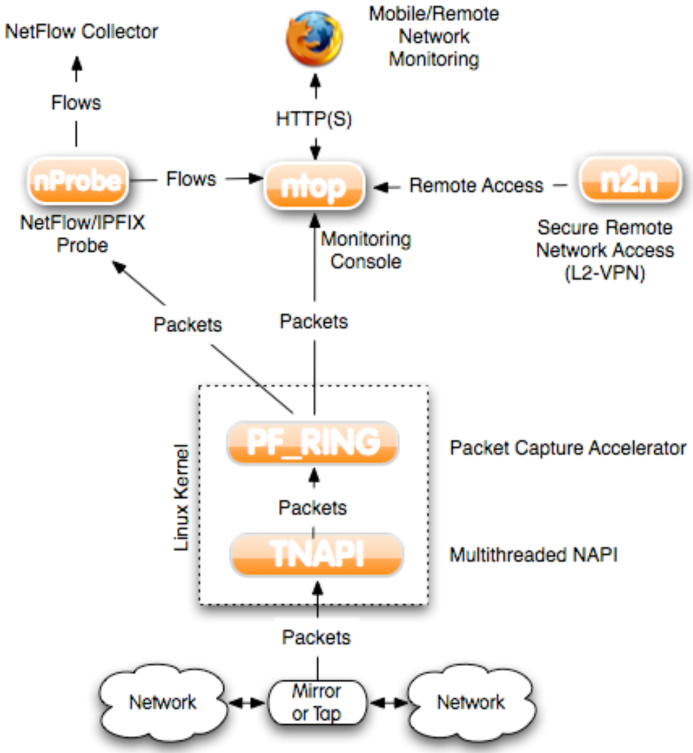
Network Traffic Monitoring

- Since the early days network monitoring has focused on packets. Indeed Wireshark is a packet analyser.
- Packets are used to deliver user data across applications.
- Users/applications have no packet visibility as they are a “low level” concept used at network layer.

(We Used to Say) Packets Never Lie

- Packet analysis provide useful information for understanding:
  - Network traffic issues.
  - Network usage not compliant with network policies (note: firewalls cannot help here).
  - Non-optimal performance.
  - Potential security flaws.
  - Ongoing (latent) attacks.
  - Data breach.

ntop Ecosystem (2009): Packets



ntop Ecosystem (2019): Still Packets

What's Wrong with Packets? [1/2]

Nothing in general, but they offer an external viewpoint

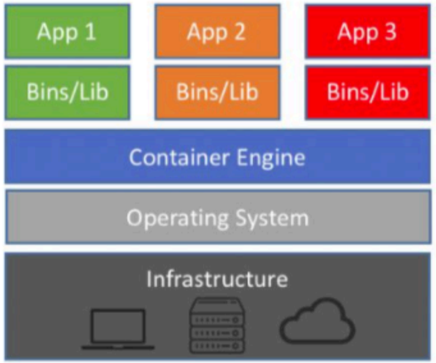
- From which we have to reconstruct what is really happening from the application/user standpoint.
- Good for monitoring network traffic from outside of a systems on a passive way (no agent installation required).
- Packets are low level and need to be "interpreted" in order to understand what happens at a higher layer: TCP zero-window, fragments, and packet retransmissions are invisible to applications and users that instead think in terms of perceived network performance and transmission errors.

What's Wrong with Packets? [2/2]

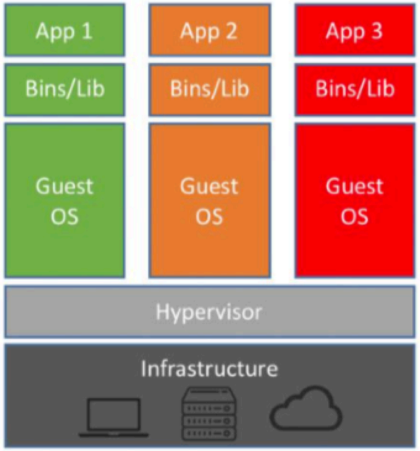
- Packets resemble synthetic information and lack of metadata that help understanding insights on the machine
- Data encryption is a challenging for DPI techniques and Wireshark, making more complicated packet payload to be dissected and decoded.
- Network administrators need to monitor packets fragmentation, flow reconstruction, packet loss/ retransmissions... metrics that would be already available inside a system but that instead are measured with packets.

What about Containers?

- Make services portable across host platforms.
- Provide an additional layer of isolation over processes.
- Allow each service to use its own dependencies.

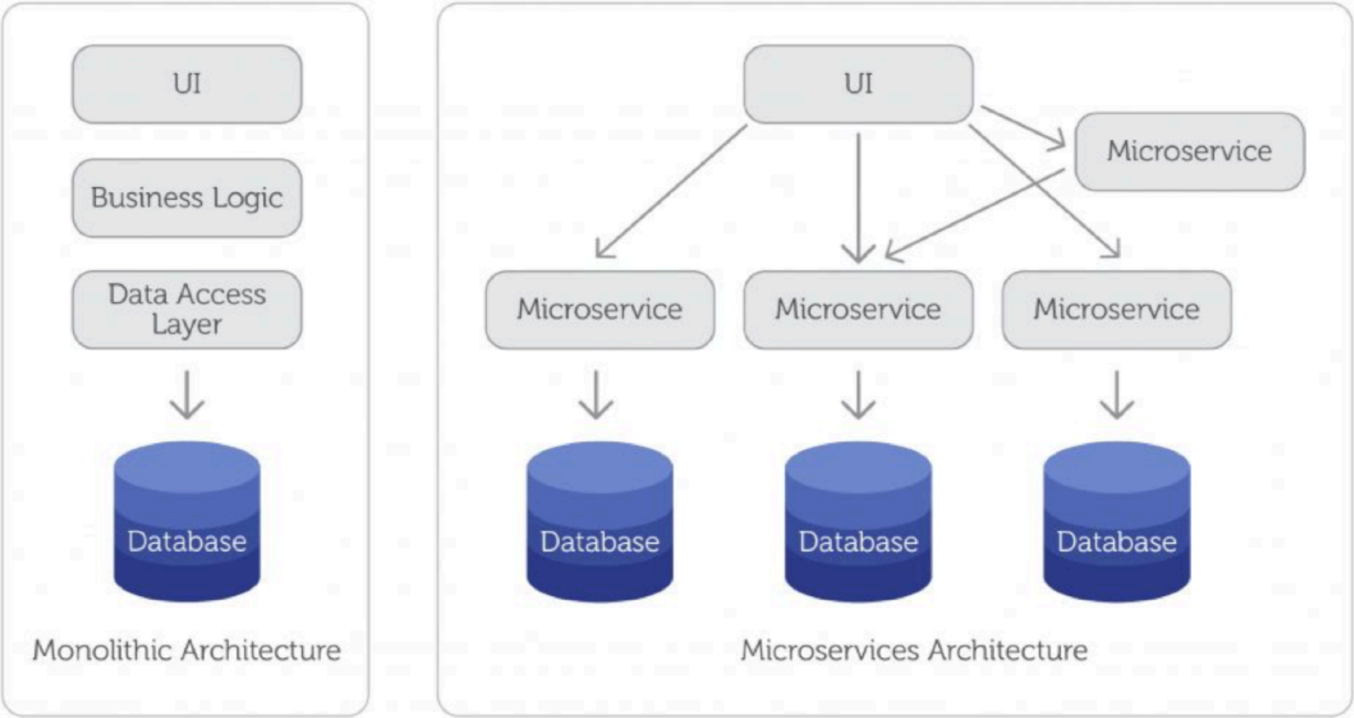


Containers



Machine Virtualization

From Monolith to Microservices [1/3]



From Monolith to Microservices [2/3]

- Code of each microservice is stored in an isolated container, runs its own memory space, and functions independently.
- Clearly organised architecture. Decoupled units have their specific jobs, can be reconfigured without affecting the entire application.
- Deployments don’t require downtime.

From Monolith to Microservices [3/3]

- If a microservice crashes, the rest of the system keeps going.
- Each microservice can be scaled individually according to its needs.
- Services can use different tech stacks (developers are free to code in any language, and HR are happy to hire programmers that do not necessarily have to code in the same programming language).

Networking and Namespaces

- In Linux network interfaces and routing tables/ entries are shared across the entire OS.
- Sometimes (containers need that) it is necessary to define different and separate instances of network interfaces and routing tables that operate independent of each other
- Linux implements this using namespaces.

Playing with Namespaces [1/4]

- Create a new namespace

```
# ip netns add wireshark

# ip netns list
cni-53bd89ab-d120-4015-0fc8-f5cb5ed45413
cni-f4c00b32-2487-8e9c-3f60-e5d425aaa1d7 (id: 2)
cni-0ce982f1-b6ac-2035-9ee0-a9cd8eb8d9d6 (id: 1)
cni-920496f6-b76f-a6e0-145f-4fa315134140 (id: 0)
wireshark
```

Playing with Namespaces [2/4]

- Create a new veth interface peer

```
# ip link add veth0 type veth peer name veth1

# ip link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp0s5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group default
qlen 1000
    link/ether 00:1c:42:85:41:62 brd ff:ff:ff:ff:ff:ff
… … … …

8: veth1@veth0: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default
qlen 1000
    link/ether 5e:cb:a9:10:50:e9 brd ff:ff:ff:ff:ff:ff
9: veth0@veth1: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default
qlen 1000
    link/ether ca:b4:d6:da:7c:b0 brd ff:ff:ff:ff:ff:ff
```

Playing with Namespaces [3/4]

- Bind a veth to a namespace

```
# ip link set veth1 netns wireshark

# ip netns exec wireshark ip link list
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
8: veth1@if9: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 5e:cb:a9:10:50:e9 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

- Configure an IP address

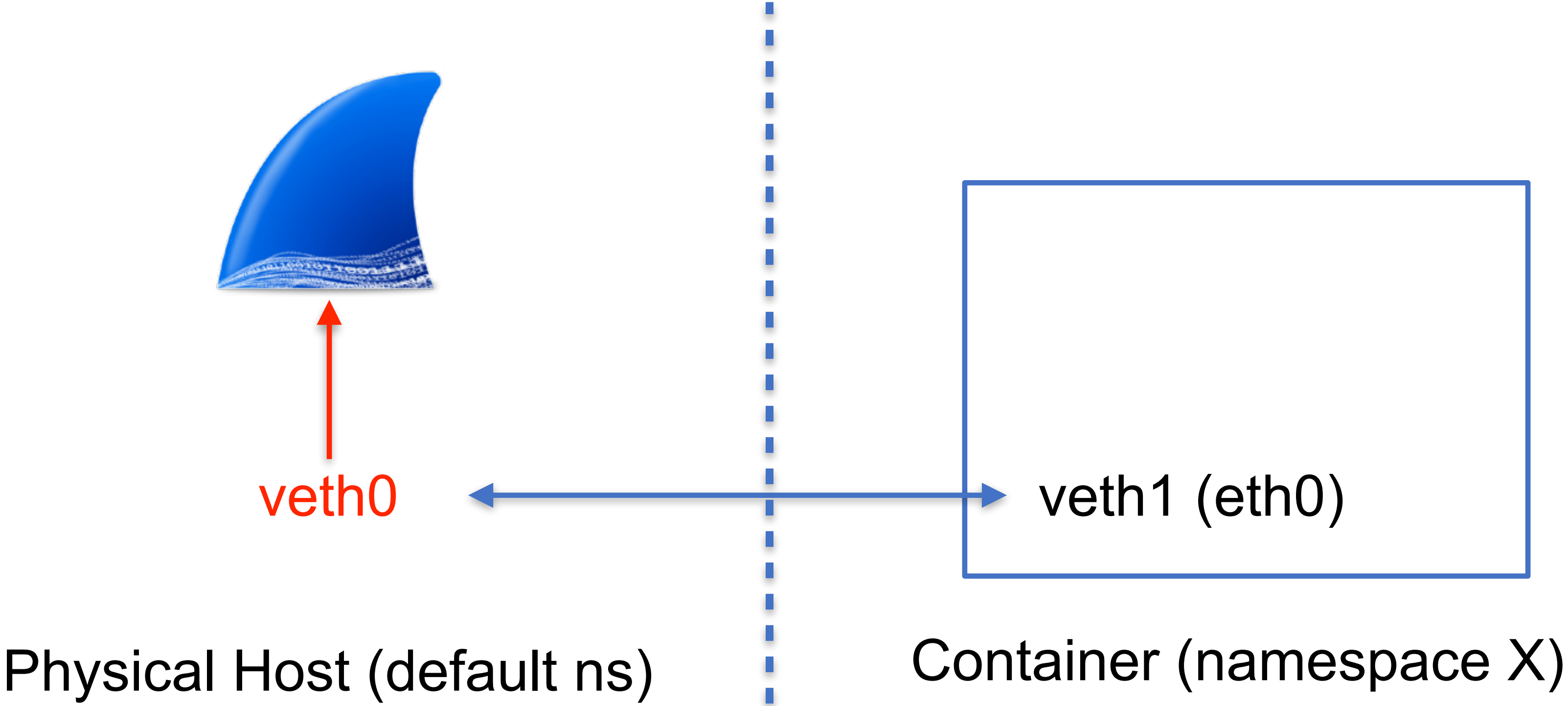
```
# ip netns exec wireshark ifconfig veth1 192.168.10.1/24 up
# ip netns exec wireshark ifconfig veth1
veth1: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 192.168.10.1  netmask 255.255.255.0  broadcast 192.168.10.255
        ether 5e:cb:a9:10:50:e9  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Playing with Namespaces [4/4]

- Adding a veth to a physical network via a bridge

```
# brctl addif cbr0 veth0

# brctl show
bridge name  bridge id          STP enabled  interfaces
cbr0     8000.6a870bb63548    no        veth0
                                        veth5a9abc1c
                                        veth884b5ab2
                                        vethc6499b6e
                                        vethd3260294
```

Namespaces and Containers



What’s Wrong with Packets on Containerised Environments? [1/2]

- Virtualisation techniques reduce visibility when monitoring network traffic as network manager are blind with respect to what happens inside the systems.
- Each container has a virtual ethernet interface so commands such as “tcpdump -i veth40297a6” won’t help as devops think in terms of container name, pod and namespace rather than veth.
- Intra-container traffic stays inside the system without hitting the wire, thus monitoring traffic from/to the host won’t help.

What’s Wrong with Packets on Containerised Environments? [2/2]

- Containers are not VMs which have a “long-range” time cycle
- Environments like Kubernetes are extremely dynamic.
- It’s hard to associate an IP address to a service because addresses have become ephemeral.
- System introspection can help us correlating the network traffic with the continuously moving parts of our infrastructure.

Network and System Visibility

- Even on a container-centric sites we still need to:
  - Monitor the infrastructure where containers are deployed: SNMP, NetFlow/IPFIX, and packets/Wireshark.
  - Enable system introspection also to (legacy) non-containerised systems so the whole infrastructure is monitored seamlessly.
- This means that we need to enable Wireshark to be used on those containerised environments.

Challenges using Wireshark with Containers [1/2]

- Intra-container traffic will never hit the wire: sniffing on eth0 won't help.
- It is not intuitive to bind a veth interface to a container name/pod in order to sniff the container traffic:
  - Each containerised environment has its own tools and naming (kubernetes != docker, Linux ns != Kubernets namespaces).
  - Interfaces appear/disappear as container are created/deleted.

Challenges using Wireshark with Containers [2/2]

- As a container pool (pod) often offers a service by load-balancing the traffic across multiple containers, it is not intuitive to follow a packet journey when passing across NAT and balancing.
- This problem will be discussed later in this presentation.

From Challenges to Solutions

- Enhance network visibility leveraging on system introspection for adding new metadata to network packets, this in order to ease troubleshooting.
- Handle virtualisation as first citizen and don't be blind (yes we want to observer containers interaction).
- Complete our monitoring journey and...
  - System Events: processes, users, containers.
  - Flows
  - Packets
- Bind metadata captured from system events at the application layer (e.g. tcp_connect invocation) to the network traffic for enabling continuous drill down: system events uncorrelated with network traffic are basically useless.

Do we still need DPI?

- In this world DPI (Deep Packet Inspection) has marginal importance since we have information on the process that generated the network event
  - User, group
  - process, absolute path, pid,
  - container id, pod, namespace
- If we are able to know that an application generated a network event and then we are able to bind that information to the network traffic then DPI makes less sense.

Design Goals

- Extend Wireshark to take into account system events in order to provide some context (process, user, PID…) to the captured traffic.
- Hide Wireshark the complexity of containerised environments and let network administrators focus on packet analysis without them being container experts.
- IMPORTANT: We **don’t want to replace** packet capture with events but **rather complement** captured traffic with **additional information**.

Early Experiments: Sysdig

- Provides a way to observe the system at the kernel system call level.
- ntop has been an early sysdig adopter adding in 2014 sysdig events support in ntop tools.
- Despite all our efforts, this activity has NOT been a success for many reasons:
  - Too much CPU load (in average +10-20% CPU load)
  - requires a new kernel module that sometimes is not what sysadmins like as it might invalidate distro support.
  - Containers were not so popular in 2014, and many people did not consider system visibility so important at that time.

How Sysdig Works

- As sysdig focuses on system calls for tracking a TCP connections we need to:
  - Discard all nonTCP related events (sockets are used for other activities on Linux such as Unix sockets)
  - Track socket() and remember the socketId to process/thread.
  - Track connect() and accept() and remember the TCP peers/ports.
  - Collect packets and bind each of them to a flow (i.e. this is packet capture again, using sysdig instead of libpcap).

- This explains the CPU load, complexity...

Using Sysdig [1/2]

```
$ curl https://www.ntop.org
```

```
# sysdig -pc evt.type=connect or evt.type=bind
25395 16:56:35.648903745 0 host (host) curl (26431:26431) > connect fd=3(<u>)
25396 16:56:35.648914011 0 host (host) curl (26431:26431) < connect res=-2(ENOENT) tuple=0->ffff9458c020ec00 /var/run/nscd/socket
25401 16:56:35.648922620 0 host (host) curl (26431:26431) > connect fd=3(<u>)
25402 16:56:35.648924967 0 host (host) curl (26431:26431) < connect res=-2(ENOENT) tuple=0->ffff9458c020ec00 /var/run/nscd/socket
25537 16:56:35.649282362 0 host (host) curl (26431:26431) > connect fd=3(<4>)
25538 16:56:35.649289899 0 host (host) curl (26431:26431) < connect res=0 tuple=131.114.21.11:42026->131.114.21.6:53
25699 16:56:35.650580211 0 host (host) curl (26431:26431) > bind fd=3(<n>)
25700 16:56:35.650582767 0 host (host) curl (26431:26431) < bind res=0 addr=NULL
25721 16:56:35.650631926 0 host (host) curl (26431:26431) > connect fd=3(<6>)
25724 16:56:35.650642514 0 host (host) curl (26431:26431) < connect res=0 tuple=2a00:d40:1:3:131.114.21:11:41764-
>2a03:b0c0:2:d0::360:4001:443
25727 16:56:35.650645184 0 host (host) curl (26431:26431) > connect fd=3(<6>2a00:d40:1:3:131.114.21:41764-
>2a03:b0c0:2:d0::360:4001:443)
25728 16:56:35.650645950 0 host (host) curl (26431:26431) < connect res=0 tuple=0.0.0.0:0->0.0.0.0:0
25729 16:56:35.650646881 0 host (host) curl (26431:26431) > connect fd=3(<4u>0.0.0.0:0->0.0.0.0:0)
25732 16:56:35.650650936 0 host (host) curl (26431:26431) < connect res=0 tuple=::ffff:131.114.21.11:45555-
>::f87c:a283:c1a3:ffff:443
25810 16:56:35.652983176 5 host (host) curl (26430:26430) > connect fd=3(<6>)
25811 16:56:35.653036637 5 host (host) curl (26430:26430) < connect res=-115(EINPROGRESS) tuple=2a00:d40:1:3:131.114.21:11:60894-
>2a03:b0c0:2:d0::360:4001:443
```

Using Sysdig [2/2]

```
$ top | grep -E 'PID|sysdig|ebpflowexport
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
25197 root      20   0  351116  16404  11552 S   0,7   0,2   0:01.49 sysdig
25133 root      20   0  159668 122128  48100 S   0,3   1,5   0:03.74 ebpflowexport
25197 root      20   0  351116  16404  11552 R  11,6   0,2   0:01.84 sysdig
25133 root      20   0  159668 122128  48100 S   3,3   1,5   0:03.84 ebpflowexport
25197 root      20   0  351116  16404  11552 S  26,7   0,2   0:02.65 sysdig
25133 root      20   0  159668 122128  48100 S   0,3   1,5   0:03.85 ebpflowexport
25133 root      20   0  159668 122128  48100 S   0,7   1,5   0:03.87 ebpflowexport
```

Load matters in particular on the cloud

Towards eBPF

- 1992: Steve McCane and Van Jacobson introduced a VM model packets filtering. This version of BPF is now known as classic BPF (cBPF)
- 1997: cBPF was introduced in Linux in kernel 2.1.75, as a technology for in-kernel packet filtering
- 2013: eBPF, created by Alexei Starovoitov, extended what bpf virtual machine could do. The VM is now able to intercept other kind of events and take several action other than filtering (https://lkml.org/lkml/2013/12/2/1066)

(ebpf official logo)

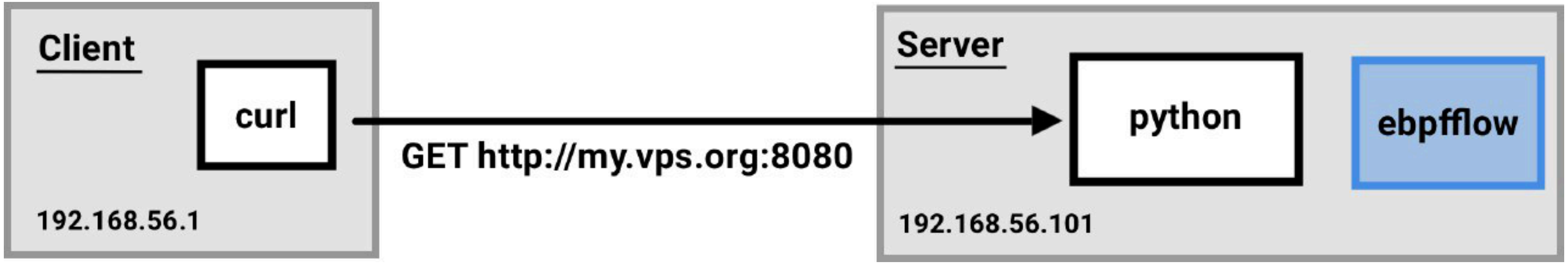
Welcome eBPF

- eBPF is great news for Wireshark as:
- It gives the ability to avoid sending everything to user-space but perform in kernel computations and send metrics to user-space.
- We can track more than system calls (i.e. be notified when there is a transmission on a TCP connection without analyzing packets).
- It is part of modern Linux systems (i.e. no kernel module needed).

libebpflow: eBPF for System Visibility

- Our aim has been to create an open-source library that offers a simple way to interact with eBPF network events in a transparent way.
    - Reliable and trustworthy information on the status of the system when events take place.
    - Low overhead event-based monitoring
    - Information on users, network statistics, containers and processes
    - Go and C/C++ support
- https://github.com/ntop/libebpfflow (GNU LGPL)

libebpflow: client-server [1/2]



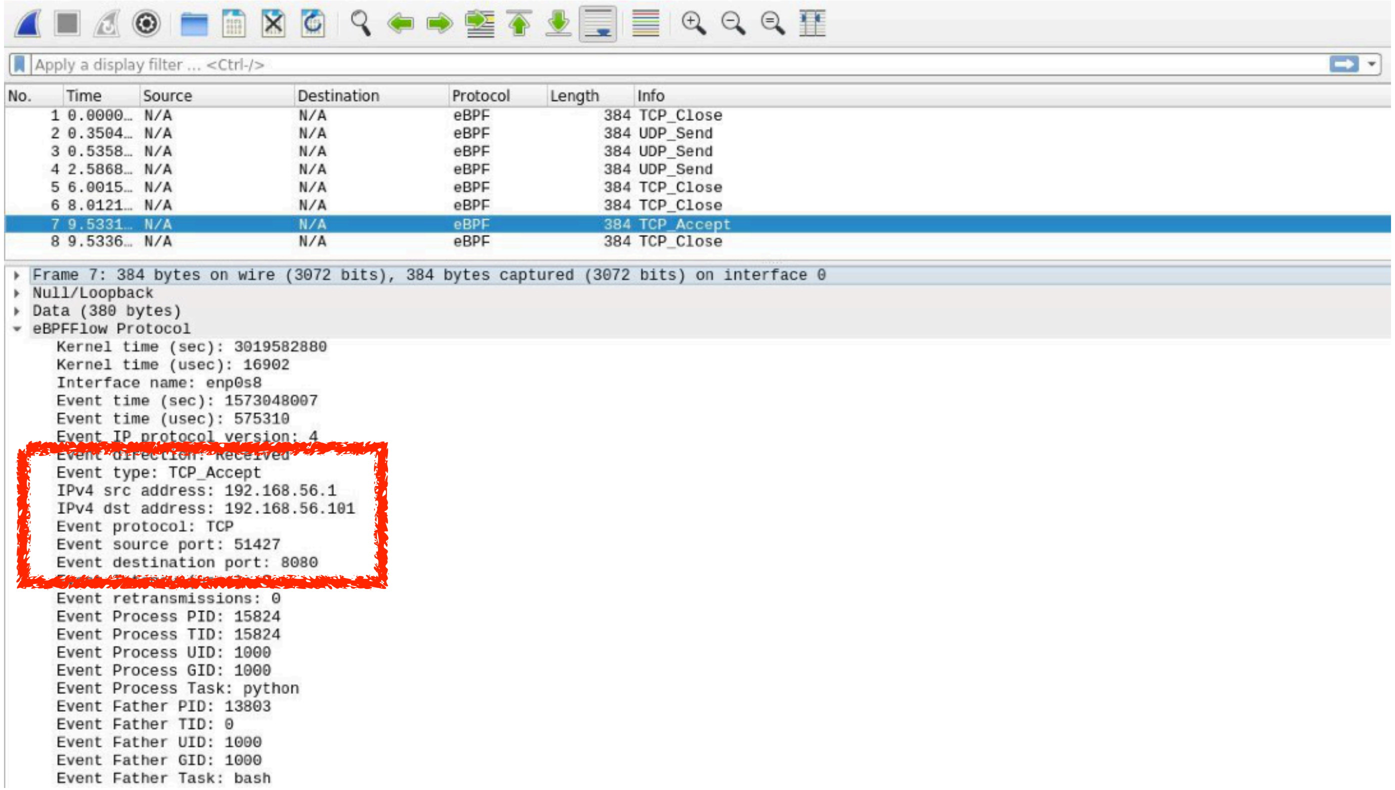
- We host a service on port 8080

  user@Server:~/$ python -m SimpleHTTPServer 8080

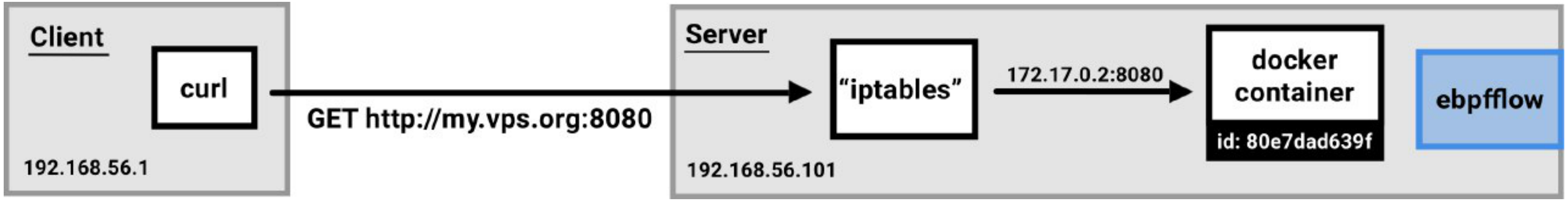
- We use curl to http-GET using the local port 1234

  user@Client::~/$ curl --local-port 1234 http://my.vps.org:8080

libebpflow: client-server [2/2]



libebpflow: client-container [1/2]



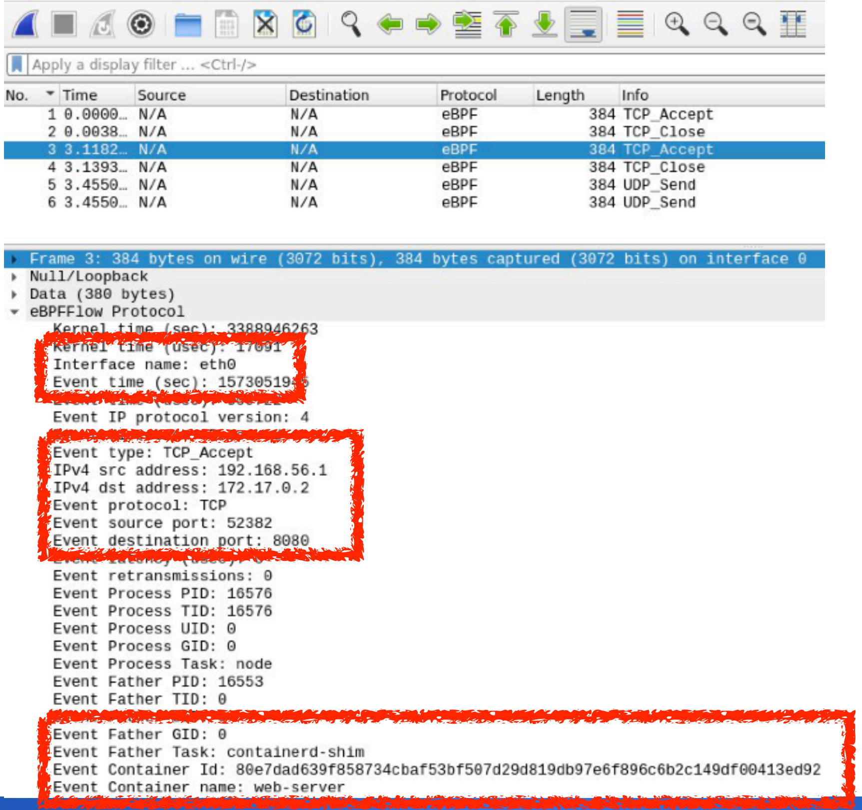
- We Run detached container which serves https on port 80

  user@Server:~/$ docker run --rm -it -p 4443:8080 sabellas/cowserve

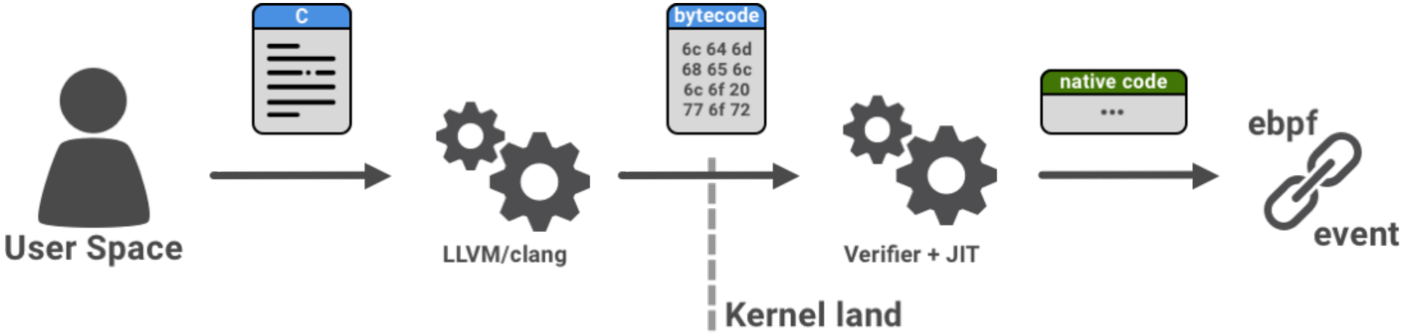
- We use curl to https-GET using the local port 1234

  user@Client:~/$ curl --local-port 1234 http://my.vps.org:8080

libebpflow: client-container [2/2]

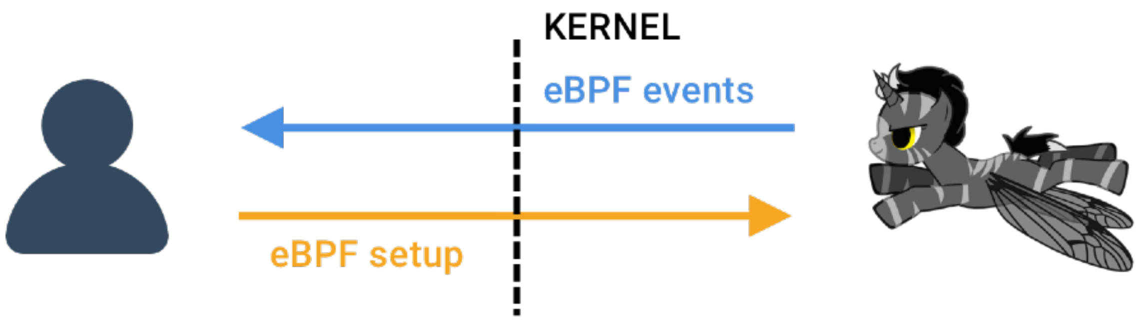


Under the Hood



- The user writes a program in C
- The program is translated in eBPF instructions (LLVM/clang)
- A verifier check if the eBPF program is safe (e.g. no loops, only known external function allowed)
- A just in time compiler translate the bytecode directly into a target architecture: x86, ARM, MIPS, etc.
- The program is attached to the target kernel event such that whenever the event is triggered the program is executed

libebpfflow Overview [1/2]

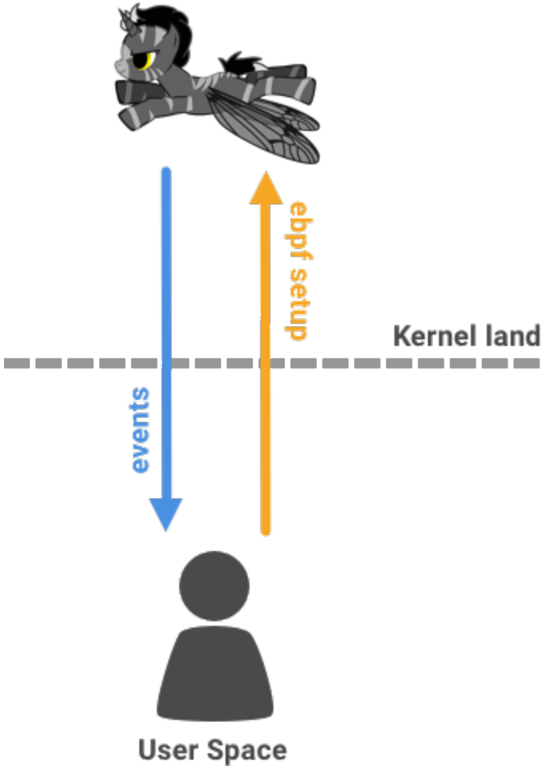



```
// Attaching probes ----- //
if (userarg_eoutput && userarg_tcp) {
    // IPv4
    AttachWrapper(&ebpf_kernel, "tcp_v4_connect",  "trace_connect_entry",     BPF_PROBE_ENTRY);
    AttachWrapper(&ebpf_kernel, "tcp_v4_connect",  "trace_connect_v4_return", BPF_PROBE_RETURN);
    // IPv6
    AttachWrapper(&ebpf_kernel, "tcp_v6_connect",  "trace_connect_entry",     BPF_PROBE_ENTRY);
    AttachWrapper(&ebpf_kernel, "tcp_v6_connect",  "trace_connect_v6_return", BPF_PROBE_RETURN);
}
```

libebpfflow Overview [2/2]

```c
typedef enum {
  eTCP_ACPT = 100,
  eTCP_CONN = 101,
  eTCP_CONN_FAIL = 500,
  eUDP_RECV = 210,
  eUDP_SEND = 211,
  eTCP_RETR = 200,
  eTCP_CLOSE = 300,
} event_type;

struct taskInfo {
  u32 pid; /* Process Id */
  u32 tid; /* Thread Id  */
  u32 uid; /* User Id    */
  u32 gid; /* Group Id   */
  char task[COMMAND_LEN], *full_task_path;
};

// separate data structs for ipv4 and ipv6
struct ipv4_addr_t {
  u64 saddr;
  u64 daddr;
};

struct ipv6_addr_t {
  unsigned __int128 saddr;
  unsigned __int128 daddr;
};
```

```c
typedef struct {
  ktime_t ktime;
  char ifname[IFNAMSIZ];
  struct timeval event_time;
  u_int8_t ip_version, sent_packet;
  u16 etype;

  union {
    struct ipv4_addr_t v4;
    struct ipv6_addr_t v6;
  } addr;

  u8  proto;
  u16 sport, dport;
  u32 latency_usec;
  u16 retransmissions;

  struct taskInfo proc, father;
  char
container_id[CONTAINER_ID_LEN];

  struct {
    char *name;
  } docker;

  struct {
    char *name;
    char *pod;
    char *ns;
  } kube;
} eBPFevent;
```

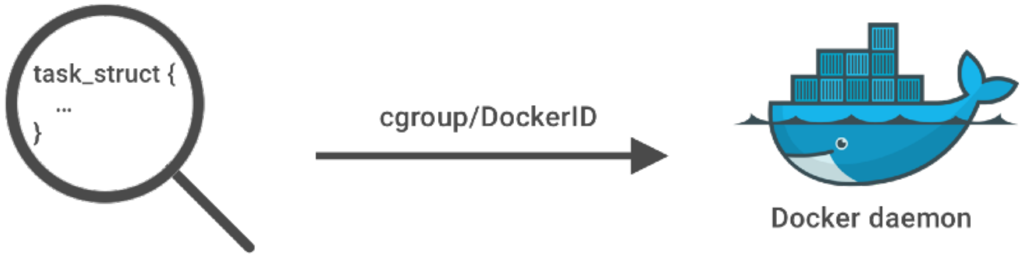


Collecting Information: Processes

- In linux every task has associated a struct (i.e. task_struct) that can be retrieved by invoking the function bpf_get_current_task provided by eBPF. By navigating through the kernel structures it can be gathered:
  - uid, gid, pid, tid, process name and executable path
  - cgroups associated with the task.
- Connection details instead are read from the socket structure. They include: source and destination ip/port, bytes send and received, protocol used.

Collecting Information: Containers

- Container can be found in proc/cgroup, however retrieving information from there is a too slow operation.
- Because containers are processes, we can navigate through kernel data structures and read information from inside the kernel where the container identifier can be collected.
- Further interaction with the container runtimes (e.g. containerd or dockerd) in use is required to collect detailed information

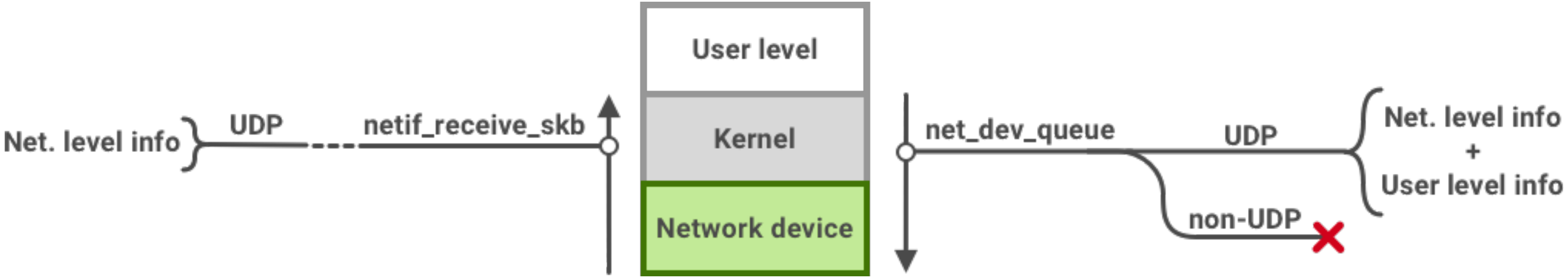


Event handling: TCP

```
inet_csk_accept (struct sock *sk, int flags, int *err)

accept
execution

ebpf event
        struct sock *newsk = (struct sock *)PT_REGS_RC(ctx);

        if(newsk != NULL) {
          eBPFevent event = { .etype = eTCP_ACPT, .ip_version = 4 };

          fill_event(ctx, &event, newsk, NULL, 0, IPPROTO_TCP, 1);
          ebpf_events.perf_submit(ctx, &event, sizeof(eBPFevent));
        }
        return(0);

continue execution
```

```
int tcp_v4_connect (struct sock *sk)
                                        BPF_HASH(cursock, u32,
                                                 struct sock_stats);
ebpf event
        cursock.update(&tid, &s);

connect
execution

ebpf event
        int ret = PT_REGS_RC(ctx); // return value
        struct sock_stats *s;
        u32 tid = (bpf_get_current_pid_tgid() >> 32) & 0xFFFFFFFF;

        s = cursock.lookup(&tid);
        if (s == NULL)
            return(0); // missed entry

        fill_event(ctx, &event, s->sk, NULL, s->ts, IPPROTO_TCP, 0);
        cursock.delete(&tid);

continue execution
```

Event handling: UDP



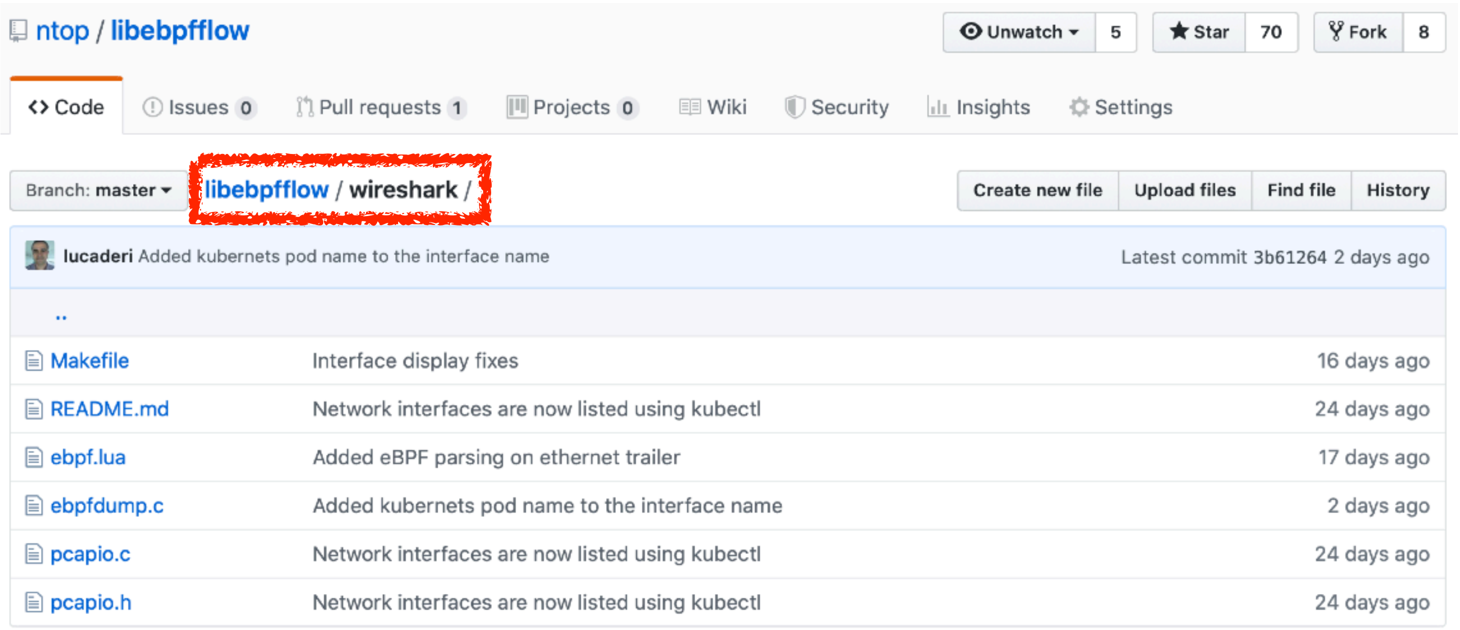
- In order to capture UDP events we attach eBPF code to net_dev_queue (process send buffer for network) and netif_receive_skb (process receive buffer from network) tracepoints and discard non UDP events.

Wireshark/libebpflow Integration

Our Original Contribution

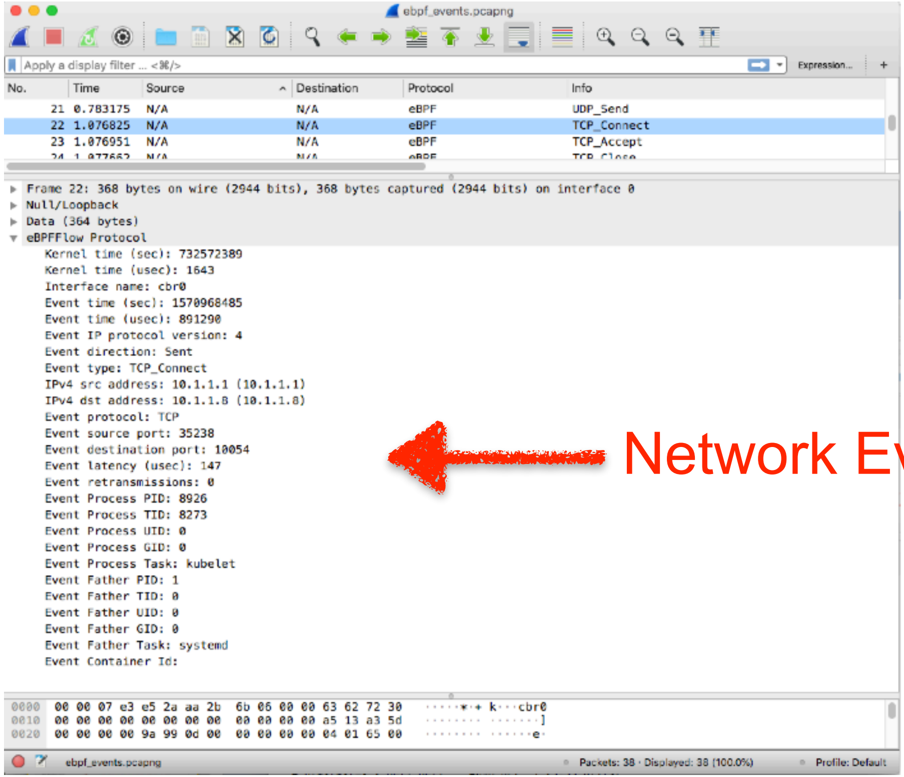
- We have developed an open source Wireshark extension that enabled network traffic monitoring by leveraging on network events. It can be used:
  - by installing a Wireshark plugin
  - from the CLI
  - or… by running a container
- The tool capture all network events within a system providing information both at
  - network level: addresses and ports
  - user level: users and processes

For the Impatient…



https://github.com/ntop/libebpfflow/tree/master/wireshark

Option 1: Events Only Monitoring



Network Events (no packets)

Option 2: Events and Packets

```
▶ Frame 207: 222 bytes on wire (1776 bits), 222 bytes captured (1776 bits)
▶ Ethernet II, Src: 82:aa:2a:6b:ed:a3 (82:aa:2a:6b:ed:a3), Dst: 52:47:27:a5:5b:41 (52:47:27:a5:5b:41)
▶ Internet Protocol Version 4, Src: 10.1.1.1 (10.1.1.1), Dst: 10.1.1.6 (10.1.1.6)
▶ Transmission Control Protocol, Src Port: 34400 (34400), Dst Port: 10054 (10054), Seq: 3477950944, Ack: 2160430723, Len: 118
▼ Hypertext Transfer Protocol
  ▶ GET /metrics HTTP/1.1\r\n
    Host: 10.1.1.6:10054\r\n
    User-Agent: kube-probe/1.15\r\n
    Accept-Encoding: gzip\r\n
    Connection: close\r\n
    \r\n
    [Full request URI: http://10.1.1.6:10054/metrics]
    [HTTP request 1/1]
    Event Process PID: 20059
    Event Process TID: 3939
    Event Process UID: 0
    Event Process GID: 0
    Event Process Task: kube-pro
    Event Container Id: kube-proxy
```

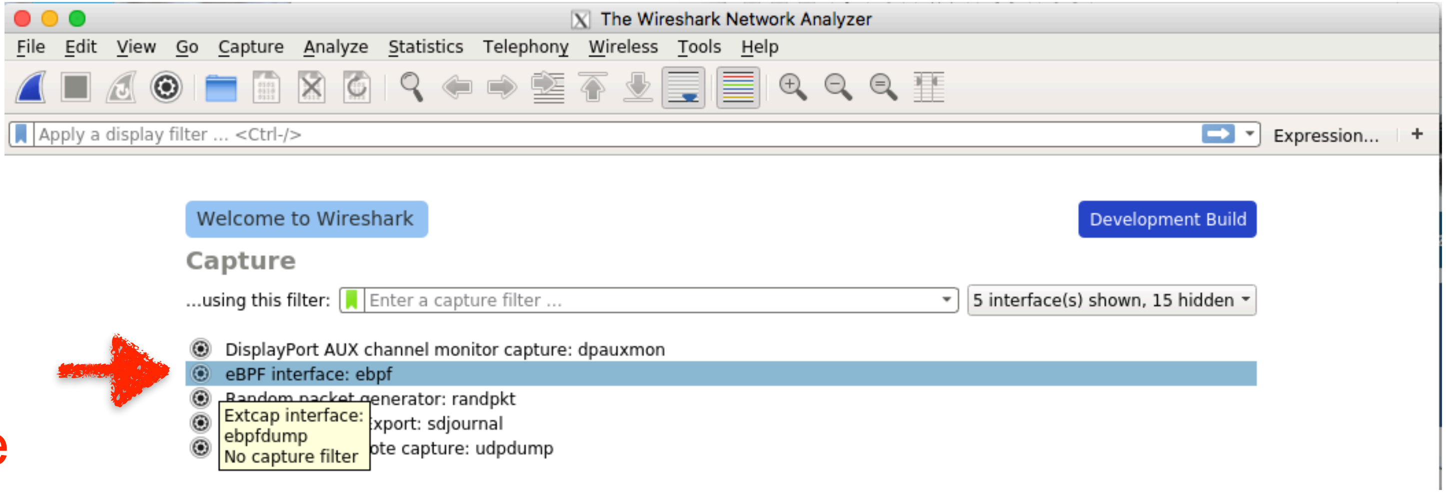
Legacy Wireshark (packets)

(Glued) Network Event

Merging Wireshark with eBPF [1/2]

- As explained before, system events are not received on a network interface but they over a kernel-to-userspace queue.
- As Wireshark is unable to handle non network-interfaces, the best solution for bringing events into it was to develop an extcap tool.

Merging Wireshark with eBPF [2/2]



Extcap
eBPF
Module

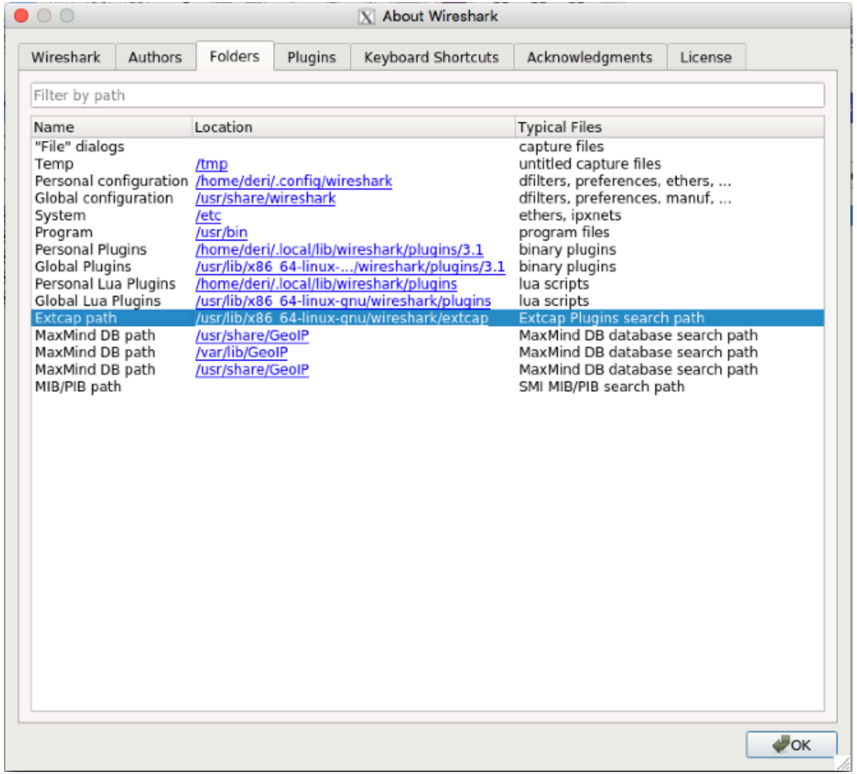
What is Extcap?

- “The extcap interface is a versatile plugin interface that allows external binaries to act as capture interfaces directly in wireshark”.
- In essence it defines a set of command line conventions to interface external tools to send wireshark captured packets (even on non-network interfaces) via a named pipe.

https://www.wireshark.org/docs/man-pages/extcap.html

ebpfdump [1/2]

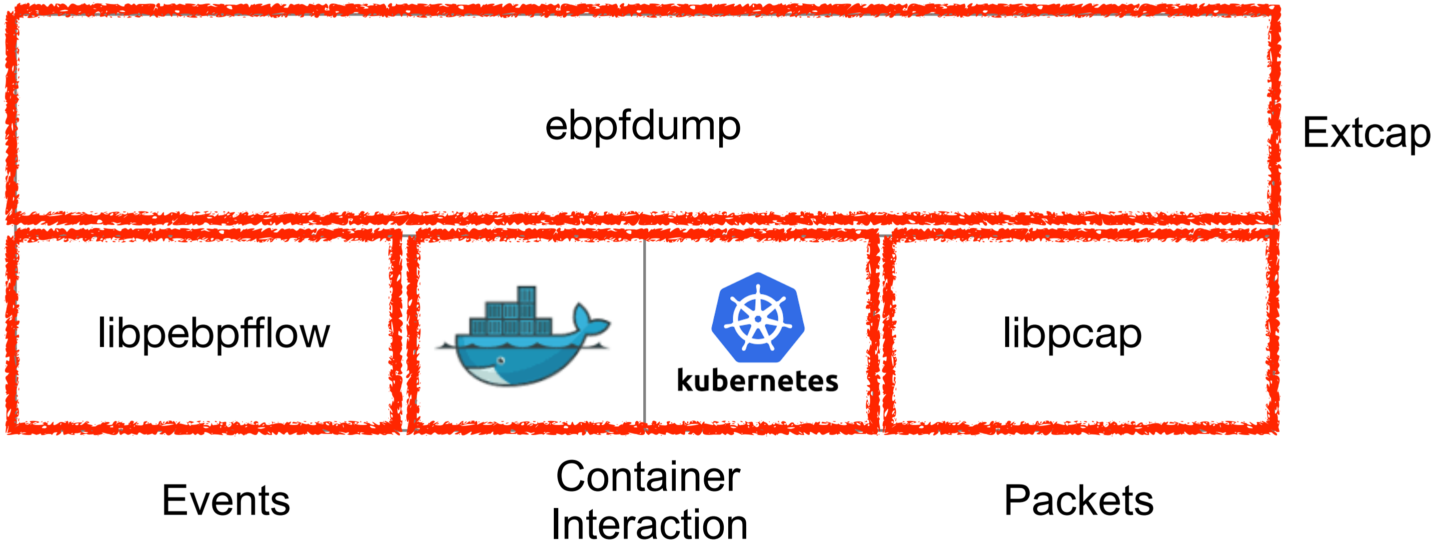
```
$ ebpfdump
Wireshark extcap eBPF plugin by ntop

Supported command line options:
--extcap-interfaces
--extcap-version
--extcap-dlts
--extcap-interface <name>
--extcap-config
--capture
--fifo <name>
--debug
--name <name>
--custom-name <name>
--help
```



ebpfdump [2/2]

```
$ ebpfdump --extcap-config --extcap-interface ebpf
arg {number=0}{call=--ifname}{display=Interface Name}{type=selector}{tooltip=Network Interface from which
packets will be captured}
value {arg=0}{value=ebpfevents}{display=eBPF Events}
value {arg=0}{value=ebpfzmqevents}{display=eBPF Remote Events (ZMQ)}
value {arg=0}{value=veth73d654ec}{display=Pod kube-dns-6bfbdd666c-5jbmx, Namespace kube-system}
value {arg=0}{value=veth02c998da}{display=Pod monitoring-influxdb-grafana-v4-78777c64c8-k8c26, Namespace
kube-system}
value {arg=0}{value=cbr0}{display=cbr0}
value {arg=0}{value=veth3b09c8fd}{display=veth3b09c8fd}
value {arg=0}{value=flannel.1}{display=flannel.1}
value {arg=0}{value=enp0s5}{display=enp0s5}
value {arg=0}{value=veth1e9ce659}{display=veth1e9ce659}
value {arg=0}{value=lo}{display=lo}
value {arg=0}{value=docker0}{display=docker0}
```

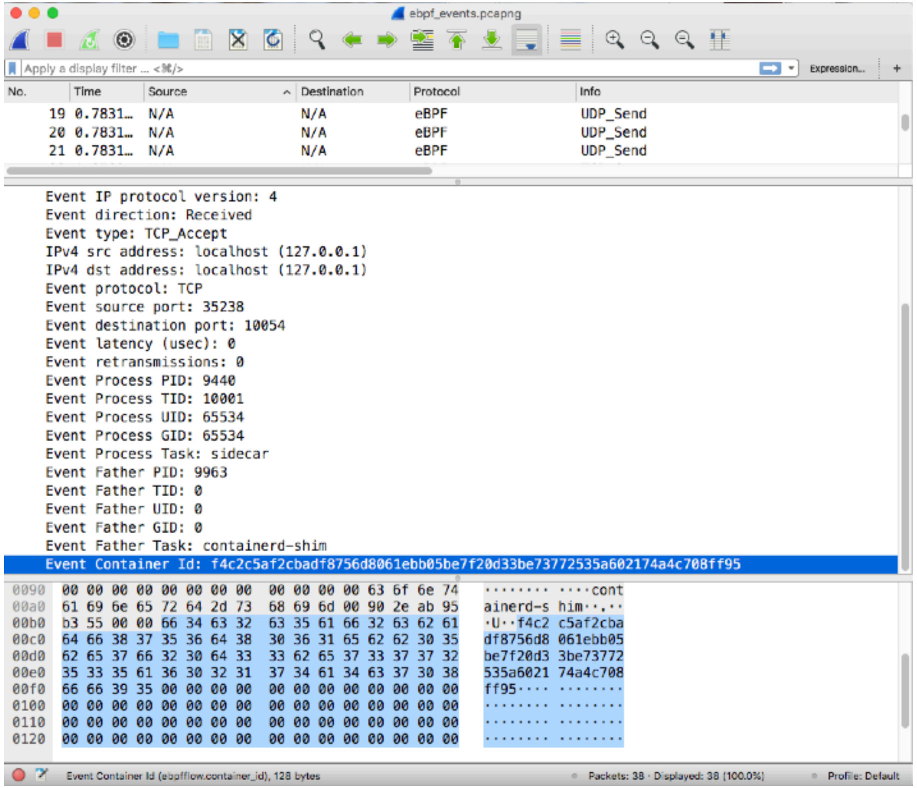
ebpfdump Architecture



ebpfdump Operating Modes [1/5]

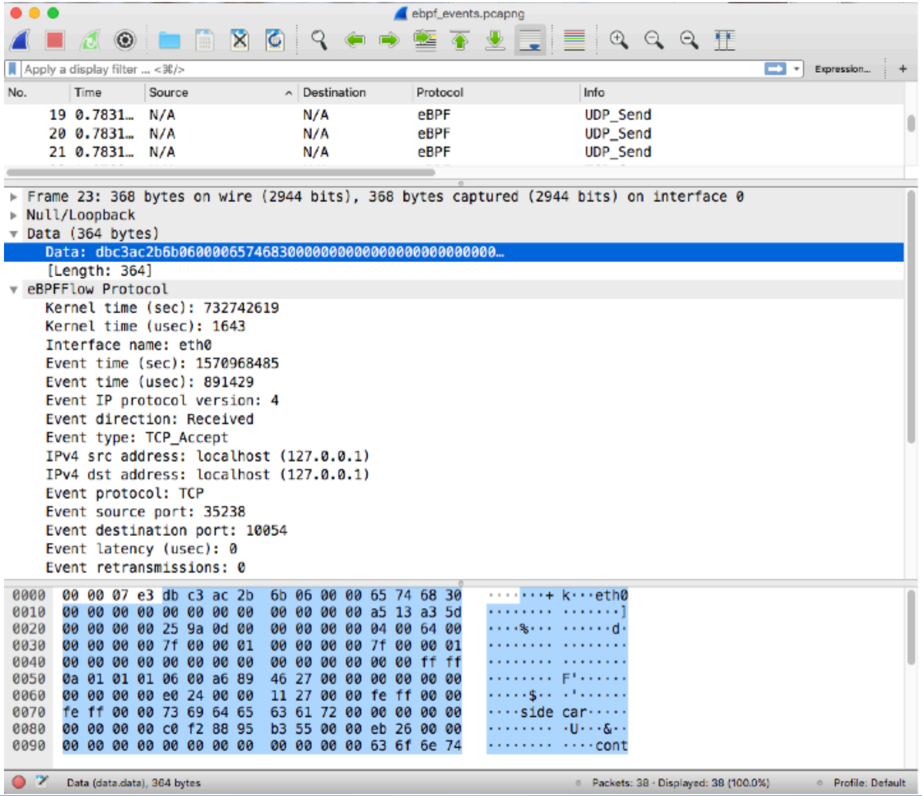
- (a) eBPF events:
  only eBPF events are returned (no packets).
  - Events are dumped as they are received and delivered to Wireshark in pcap format.
  - A lua dissector companion file decodes the received events and show them in human friendly mode.

ebpfdump Operating Modes [2/5]



ebpfdump Operating Modes [3/5]

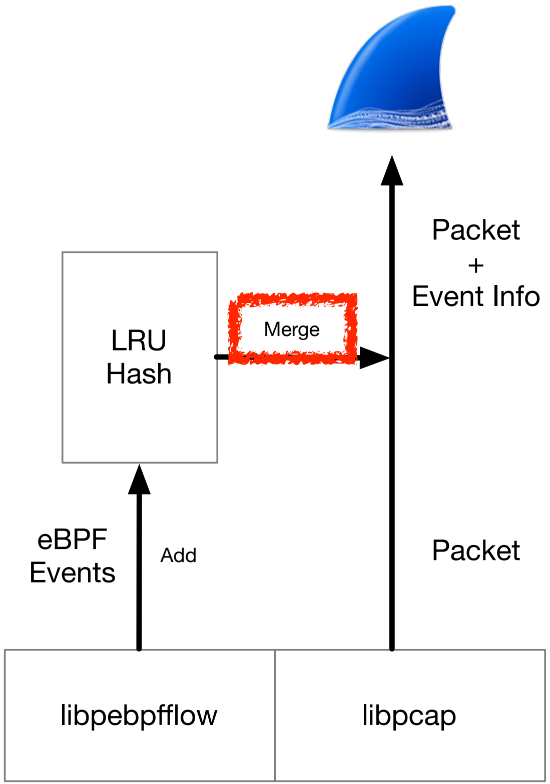
Event



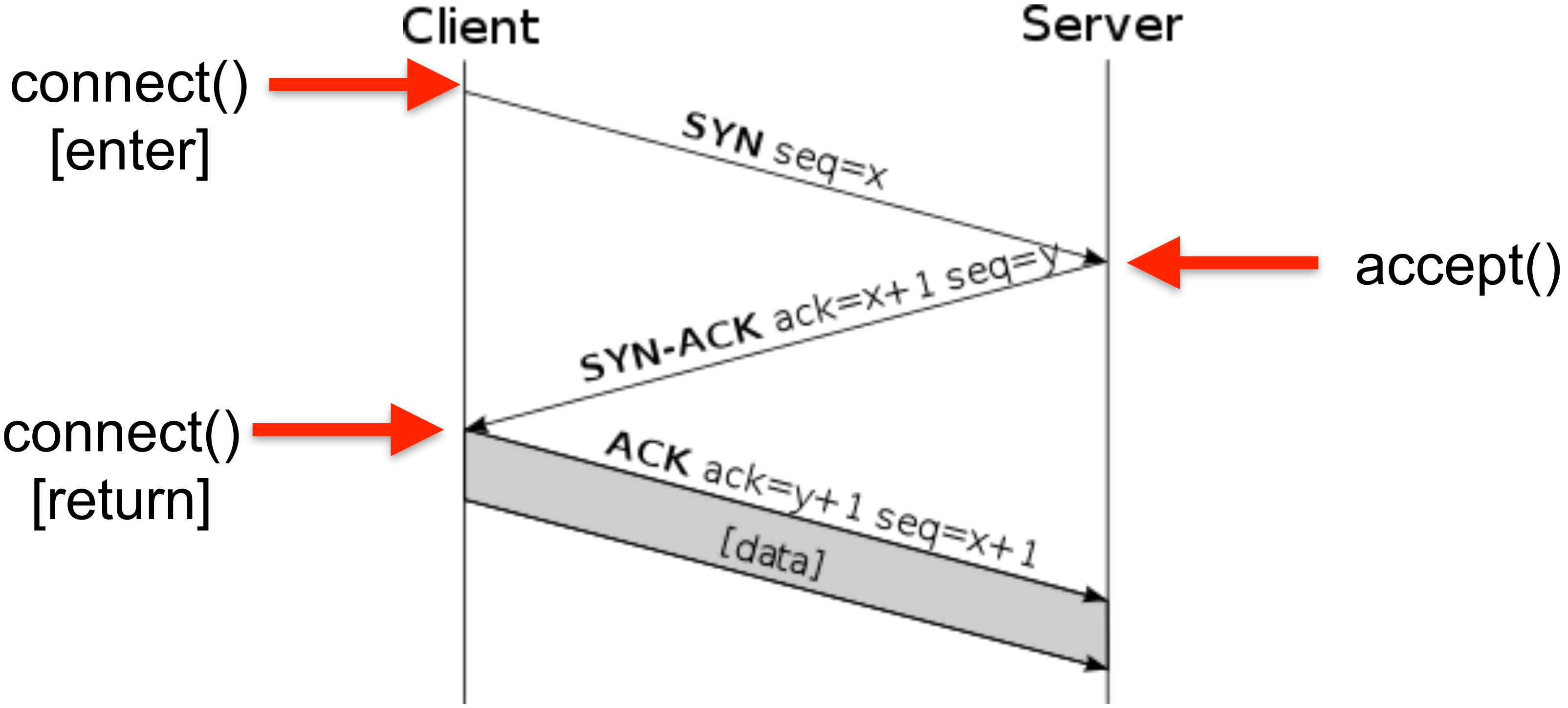
ebpfdump Operating Modes [4/5]

- (b) Packets + eBPF events.
  - Events are received and stored on a LRU hash table that will be used to match packets.
  - Received packets are matched against the LRU hash table and in case of a match, the packet is extended to add event information

ebpfdump Operating Modes [5/5]



Merging Events with Packets: Timing [1/3]

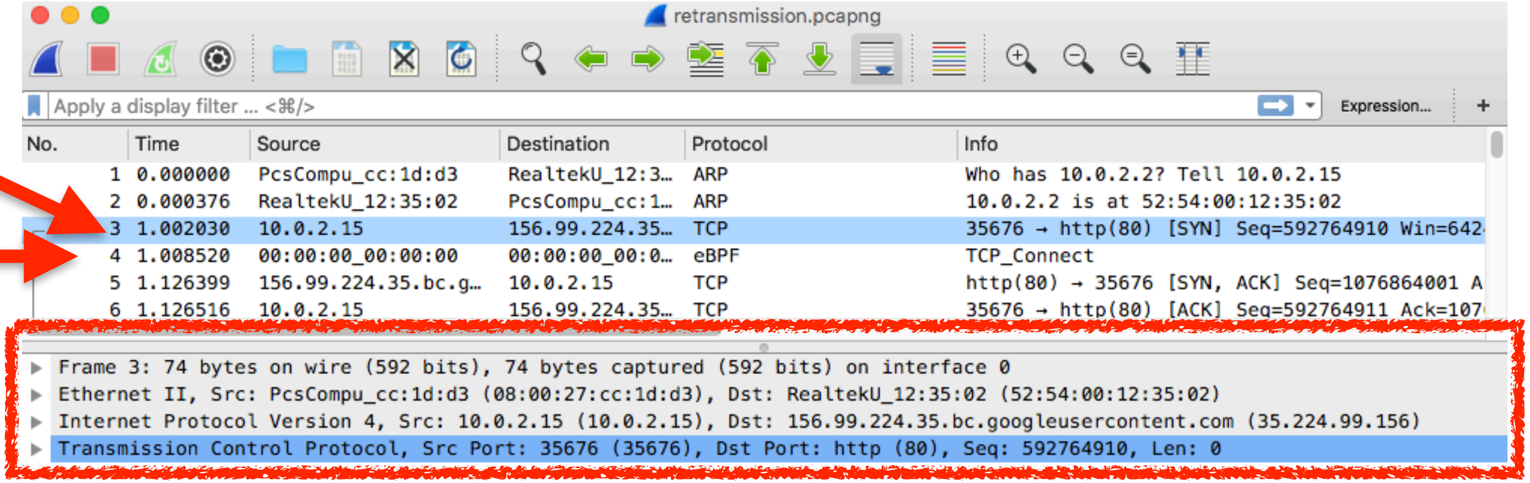


Merging Events with Packets: Timing [2/3]

SYN

Event

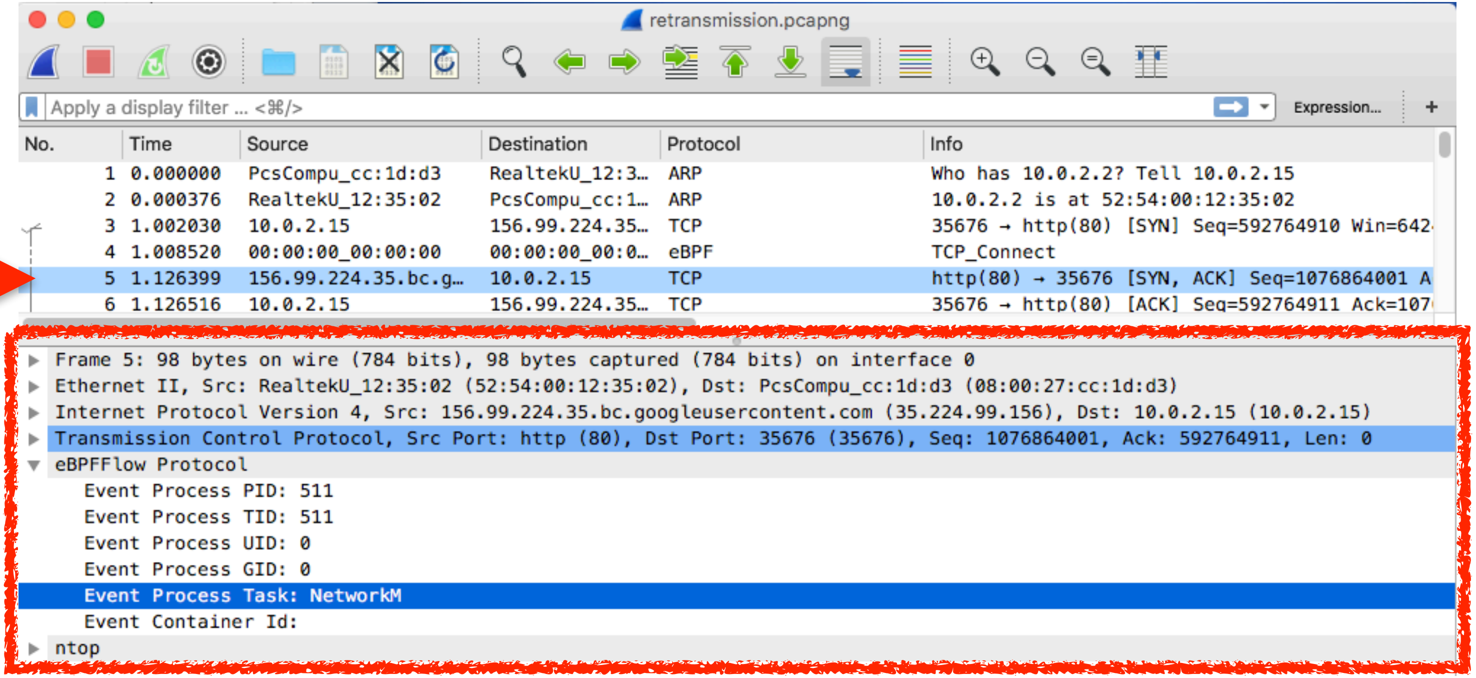
No Merge
(too early)



Merging Events with Packets: Timing [3/3]

Event

Data Merge
(Event + Pkt)



Merging Events with Packets: Naming [1/3]

- Mapping ContainerIds with Host Interfaces

```
root@ntop-ubuntu:/home/deri/libebpfflow/utils# ./docker_show_veth.sh
veth        containerId
-----------------------
vethd38ebdb xenodochial_rosalind

root@ntop-ubuntu:/home/deri/libebpfflow/utils# ifconfig vethd38ebdb
vethd38ebdb: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::9803:15ff:fe41:5b47  prefixlen 64  scopeid 0x20<li
        ether 9a:03:15:41:5b:47  txqueuelen 0  (Ethernet)
        RX packets 65  bytes 5844 (5.8 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 127  bytes 11706 (11.7 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

eBPF Events
Container xenodochial_rosalind
docker0
cbr0
vethd38ebdb
flannel.1
veth215c2ad2
enp0s5
veth67e78b7a
veth3db12b7c
veth24b4614d
lo

Merging Events with Packets: Naming [2/3]

Container Interface (no vethXXX, won't help)

```
1570482464.998115 [eth0][Rcvd][IPv4/TCP][pid/tid: 17330/17330 [/usr/bin/python2.7], uid/gid: 0/0][father pid/tid: 17158/0 [/bin/bash], uid/gid: 0/0][addr: 192.168.1.202:54235 <-> 172.17.0.2:80][ACCEPT][containerID: 79ba73e1213768da608fca002c6b2f5b0c994ce3c4cf62acf1805ebef293b418][docker_name: xenodochial_rosalind]
```

No.	Time	Source	Destination	Protocol	Length	Info
3	0.000048	172.17.0.2	192.168.1.202	TCP	112	80 → 54235 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=146
4	0.000282	192.168.1.202	172.17.0.2	TCP	104	54235 → 80 [ACK] Seq=1 Ack=1 Win=131744 Len=0 TSval=69267
5	0.000750	192.168.1.202	172.17.0.2	HTTP	186	GET / HTTP/1.1
6	0.000769	172.17.0.2	192.168.1.202	TCP	104	80 → 54235 [ACK] Seq=1 Ack=83 Win=29056 Len=0 TSval=31558
7	0.001367	172.17.0.2	192.168.1.202	TCP	121	80 → 54235 [PSH, ACK] Seq=1 Ack=83 Win=29056 Len=17 TSval
8	0.001479	192.168.1.202	172.17.0.2	TCP	104	54235 → 80 [ACK] Seq=83 Ack=18 Win=131744 Len=0 TSval=692
9	0.001516	172.17.0.2	192.168.1.202	TCP	141	80 → 54235 [PSH, ACK] Seq=18 Ack=83 Win=29056 Len=37 TSva

- Frame 3: 112 bytes on wire (896 bits), 112 bytes captured (896 bits) on interface /tmp/wireshark_extcap_ebpf_20191007230738_PQ2vjp, id 0
- Ethernet II, Src: 02:42:ac:11:00:02 (02:42:ac:11:00:02), Dst: 02:42:9d:03:10:a8 (02:42:9d:03:10:a8)
- Internet Protocol Version 4, Src: 172.17.0.2, Dst: 192.168.1.202
- Transmission Control Protocol, Src Port: 80, Dst Port: 54235, Seq: 0, Ack: 1, Len: 0
- eBPFFlow Protocol

Merging Events with Packets: Naming [3/3]

- Merging Via Container Name
  - Container Name (Docker)

```
root      11334 58.6  5.2 232640 106200 pts/1   S    23:16   0:03 /usr/
lib/x86_64-linux-gnu/wireshark/extcap/ebpfdump --capture --extcap-
interface ebpf --fifo /tmp/wireshark_extcap_ebpf_20191007231612_01Q40m --
ifname vethd38ebdb@xenodochial_rosalind
```

  - Pod (Kubernetes)

```
/usr/lib/x86_64-linux-gnu/wireshark/extcap/ebpfdump --capture --extcap-interface
ebpf --fifo /tmp/wireshark_extcap_ebpf_20191007234339_IIdYnh --ifname
veth24b4614d@kube-dns-6bfbdd666c-5jbmx
```

Merging Events with Packets: Headers [1/4]

- Start the container (container eth0 172.17.0.2)

$ docker run -d --name=Jupyter_Test --rm -p 4443:8888 jupyter/datascience-notebook

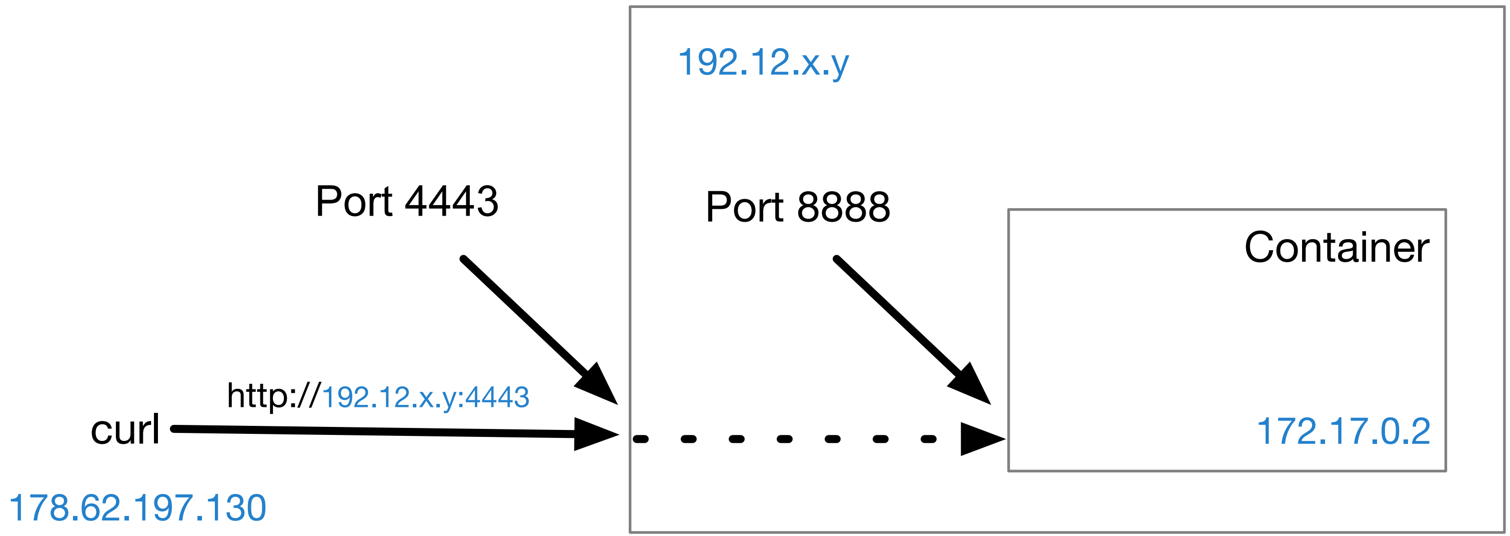
- Connect from remote

curl http://host_ip:4443

1570441451.556130 [eth0][Rcvd][IPv4/TCP][pid/tid: 10713/10713 [/opt/conda/bin/python3.7], uid/gid: 1000/100][father pid/tid: 10594/0 [/opt/conda/bin/tini], uid/gid: 1000/100][addr: 178.62.197.130:60905 <-> 172.17.0.2:8888][ACCEPT][containerID: e6296af6[illegible]a71795c60ff6d5034834ed[illegible]216b598658a7111cad42a5de9ffe67ee][docker_name: Jupyter_Test]

Remote IP — Container IP — Host Mapped Port — Container Local Port

Merging Events with Packets: Headers [2/4]



Merging Events with Packets: Headers [3/4]

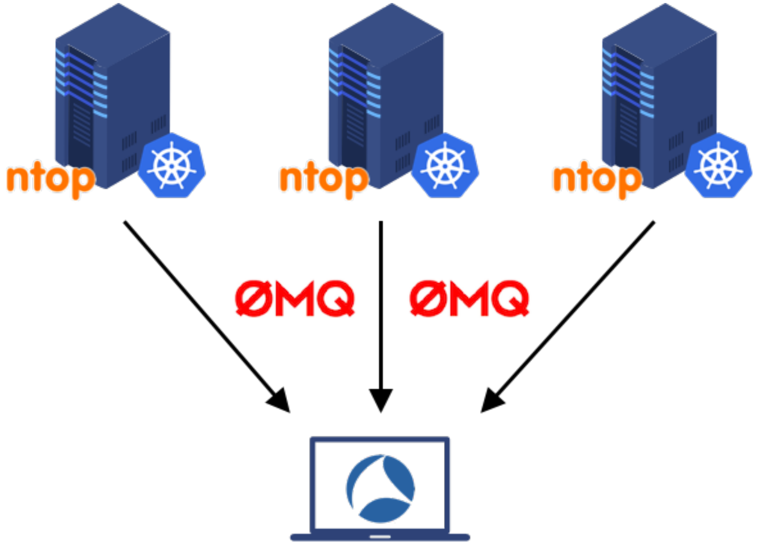
- Linux DNAT (Destination NAT) does the magic mapping ports and IP addresses

```
# lsof -i -n|grep 4443
docker-pr 16671          root    4u  IPv6 124484430      0t0  TCP *:4443 (LISTEN)

# iptables -L -t nat |grep 4443
DNAT       tcp  --  anywhere             anywhere             tcp dpt:4443 to:172.17.0.2:8080

# ps auxw|grep 16671
root      16671  0.0  0.0 378868  2752 ?        Sl   11:54   0:00 /usr/bin/docker-proxy -proto
tcp -host-ip 0.0.0.0 -host-port 4443 -container-ip 172.17.0.2 -container-port 8080
```

Merging Events with Packets: Headers [4/4]

- As you can see with eBPF we observe
  - Remote IP address and port
  - Container IP and local port
  - No host information reported in events (transparent to the event).
- This means that events can be mapped to packets only on vethX interfaces as on the physical host interface packets will not have the same 5-tuple of the events.

How to Capture on Multiple Hosts?

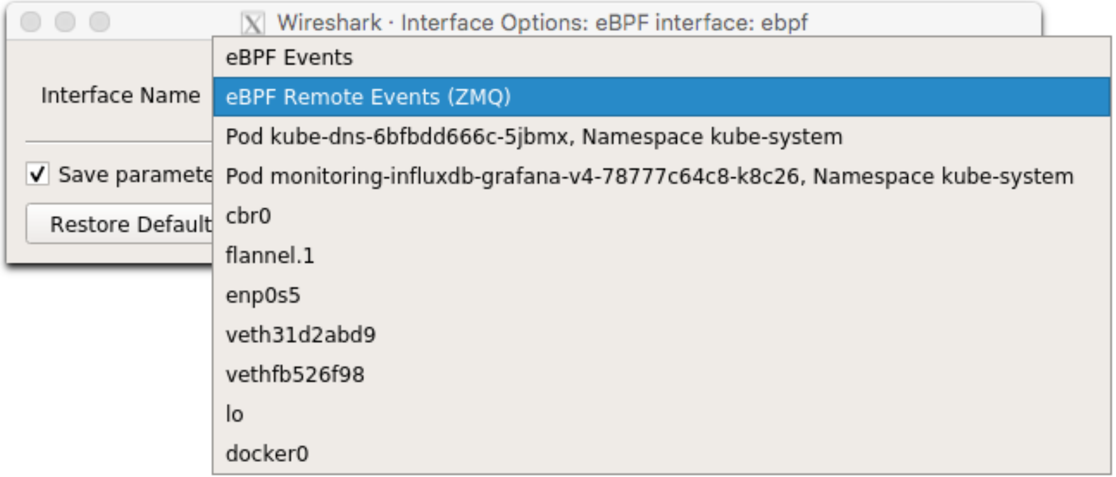


Publishers

Subscriber

Remote Flow Collection

- Enable flow collection on the host where Wireshark is running (1:N topology)

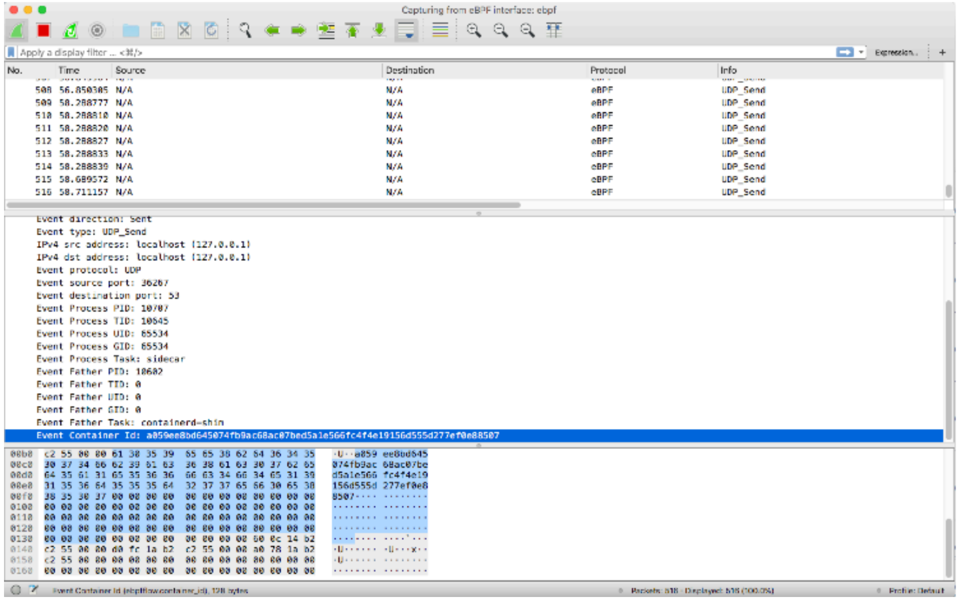


Remote Flow Export

- Each remote host needs to run
  - ebpflowexport -z “tcp://<wireshark PC>:6789c"
  - Flows are exported and sent in binary format on the “ebpf” topic.
  - The extcap plugin receives the flows and passes them to Wireshark

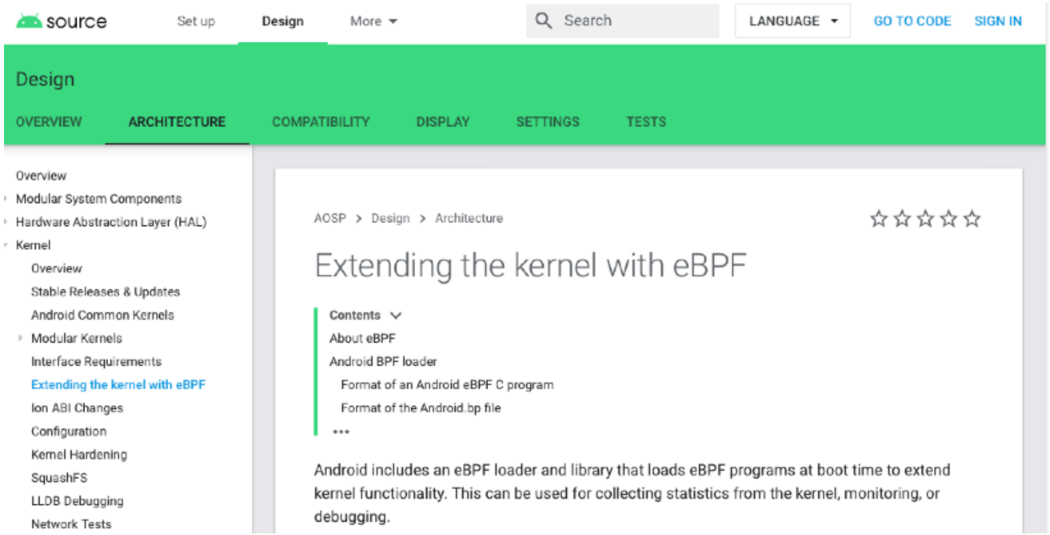
eBPF on non-Linux OSs

- ZMQ flow collection allows events to be delivered remotely
- Extcap module ported to MacOS (and potentially on other platforms such as Windows)



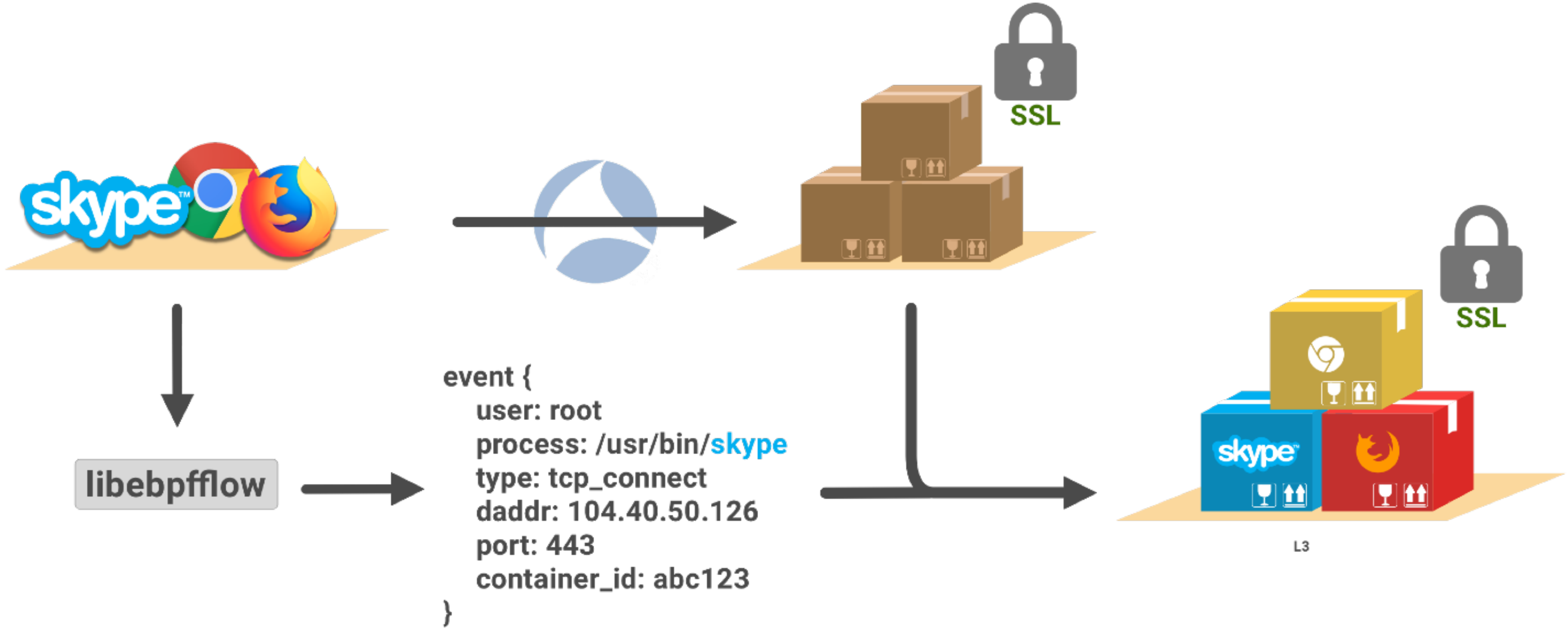
Future Work: Android

- eBPF is just being supported on Android…



https://source.android.com/devices/architecture/kernel/bpf

Wireshark+eBPF: Complete Picture



Thank You